A Layer-based Approach to Hierarchical Dynamically-scoped Open Classes

Matthias Springer†, Hidehiko Masuhara†, Robert Hirschfeld§

† Department of Mathematical and Computing Sciences / School of Computing, Tokyo Institute of Technology
§ Hasso Plattner Institute, University of Potsdam

August 10, 2016
Introduction

• ”Open Classes” in Ruby
  - Modify existing classes or modules
  - Add or overwrite methods

• Why Open Classes?
  - Object-oriented auxiliary methods
e.g.: \texttt{5.minutes + 9.hours}
  - Multi-dimensional separation of concerns [Tarr99]
  - Bug fixing (\textit{monkey patching})

• Support in programming languages
  - Ruby: open classes
  - Smalltalk: extension methods
  - Python: modifiable method dictionary
Open Classes in Ruby by example

```ruby
# in standard library
class Fixnum
  ...
end

# in a different component
class Fixnum
  def minutes
    return self * 60
  end

  def hours
    return self * 3600
  end
end
```
Example 1: WebPage Library [Takeshita13]

• A library: WebPage renders HTML and might show popups

• Two applications: using WebPage
  • Browser: should *not* show popups
  • Viewer: should show popup
The Problem: Global Visibility

class WebPage
  def open(url)
    # ...
    if popup_requested
      popup(...)
    end
  end

  def popup(text)
    # show popup window
    end
end

class Browser; end

class Viewer; end

Browser app
Viewer app

OK NG!
The Problem: Global Visibility

class WebPage
  def open(url)
    # ...
    if popup_requested
      popup(...)
    end
  end
  def popup(text)
    # show popup window
  end
end

class Browser; end
  overwrite

class Viewer; end (should show popups) NG!

Modifications are visible everywhere ↓
Other components can break
Example 1: With Open Classes

```ruby
class WebPage
  def popup(text); end
end

class Viewer
  def check(file)
    # ...
    if file.is_confidential?
      page.popup("<b>confidential</b>")
    end
  end
end
```

Open Classes are globally visible in Ruby → Viewer broken

overwrite with No-Op
Example 1: With Open Classes

```ruby
require "webpage"
require "viewer"
require "browser"

class Application
  def main
    Browser.new.open("http://www.titech.ac.jp")
    Viewer.new.check("secret.html")
  end
end
```

The diagram illustrates the relationship between the `WebPage`, `Viewer`, and `Browser` classes. The `WebPage` class has a method `popup` which is overwritten by the `Browser` class when opening a webpage.
The Problem with Open Classes

• Global Visibility
  • Modifications are visible everywhere
  • Other components (e.g., viewer) can break
    \( \rightarrow \) "Destructive Modifications"

• Solution: Locality of Changes
Our Solution: *Extension Classes*

Idea: scope control of modifications

- *Using only classes* (vs. classboxes, method shells etc.)
- Reusability through Ruby *modules* (or mixins) (vs. new syntax for refinements)
- Consistent with Ruby’s language features: take into account *class nesting hierarchy*
- Amenable to other programming languages with
  - object-based, class-based
  - unit of reuse (e.g., mixins/modules, traits, ...)
  - (class nesting hierarchy)
Extension Classes

• Modifications are defined as "inner classes"
• only visible from "enclosing classes" (details follow)

```ruby
class Browser
  def open(url)
    WebPage.new.open(url)
  end

  partial
  class ::WebPage
    def popup; end
  end
end
```

Browser app

modifies (global) WebPage but only visible from Browser
Subtleties of visibility

should modification visible when it is called:

• directly?
• from a different class?
• via a different class?
• via another method in the modified class?
• via a sibling inner class?
• via a superclass?
• via a subclass?
Our principle of visibility

Modifications are visible
• in the context of an enclosing class, and
• as long as the context remains within enclosing/sibling classes (cf. COP)

A workaround to affect external classes: empty modifications
Activation Rule

• Set of active classes $S = \{ \}$
• When calling a method $c\_.\text{foo}$: Add $c$ to $S$

```python
browser.open(...)  # S += Browser
viewer.check(...)  # S += Viewer
```
Deactivation Rule

• Restore original S when returning from a method call
• When calling C.foo, deactivate all classes a ∈ S, where C ∉ scope(a)
  • Intuitively: scope(C) is a set of classes that are compatible with the modifications defined by C
  • Mathematically: scope(C) = { C } ∪ all target classes
  • Definition will be extended later

```ruby
class Browser
  partial
  class ::WebPage; end
end
```

```ruby
scope(Browser) =
  { Browser, WebPage }
  • Browser is compatible with Browser’s modifications
  • WebPage is compatible with Browser’s modifications
```
Example 1: Overview

class Application
  def main
    Browser.new.open("http://www.titech.ac.jp")
    Viewer.new.check("secret.html")
  end
end
Example 1: Step by Step

\[
S = \{ \}
\]
Example 1: Step by Step

S = \{\text{Application}\}
Example 1: Step by Step

```plaintext
S = \{ \text{Browser} \}
```

```
scope(\text{Application})
    = \{ \text{Application} \}

\text{Browser} \notin scope(\text{Application})
```

```
\text{Application.run}
```

- \text{Application}
+ \text{Browser}

```
\text{Browser.open}
```

```
\text{Object Object (Object (WebKit, Browser, Viewer, Application))}
```

```
\text{WebKit (WebKit) popup open}
\text{Browser (browser, WebKit) open}
\text{Viewer (Viewer) check}
\text{Application (Application) main}
```
Example 1: Step by Step

\[
S = \{ \text{Browser, WebPage} \}
\]

\[
\text{scope(Browser)} = \{ \text{Browser, WebPage} \}
\]

WebPage.open

Browser.open

Application.run
Example 1: Step by Step

S = \{ \text{Browser, WebPage} \}

- **Application.run**
  - **Browser.open**
    - **WebPage.open**
      - no popup shown, because Browser is active
Example 1: Step by Step

\[
S = \{ \text{Browser, WebPage} \}
\]

Object (Object, WebPage, Browser, Viewer, Application)

WebPage (WebPage)
- popup
  - WebPage (browser, WebPage)
    - popup
      - WebPage (check)
        - main
          - do nothing

Browser.open

WebPage.open

Application.run

August 10, 2016

第110回プログラミング研究発表会 (SWoPP2016)
Example 1: Step by Step

Object {Object, WebPage, Browser, Viewer, Application}

WebPage (WebPage)
- popup
- open

Browser (browser, WebPage)
- open
- popup
- do nothing

Viewer (Viewer)
- check

Application (Application)
- main

WebPage.popup

WebPage.open

restore S

Browser.open

Application.run

S = { Browser }
Example 1: Step by Step

```
S = \{ \text{Application} \}
```

![Diagram showing the steps of an example with objects and methods such as `Application.run`, `WebPage.open`, `Browser.open`, and `WebPage.popup`.](image-url)
Example 1: Step by Step

```
S = { Viewer }
```

```
Application.run

Viewer.check
```
Example 1: Step by Step

Application.run

S = { WebPage }
Example 1: Step by Step

Application.run

S = { WebPage }

Object

WebPage

Browser

Viewer

Application

main

WebPage.open

Browser.open

WebPage.open

popup will be shown

popup will be shown
Example 1: Variations

class Application
  def main
    # ...
  end

partial
class ::WebPage
  def popup
    # ...
  end
end

class ::Browser; end
class ::Viewer; end
end

scope(Application) = 
  { Application, WebPage, Browser, Viewer}
Reusability with Modules

- Classes and modules can define partial classes
- Modifications are active in including classes (as if they were defined there directly)

```ruby
module NoPopup
  partial

  class ::WebPage
    def popup
      # ...
    end
  end
end

class Browser
  include NoPopup
  # ...
end

class Viewer
  include NoPopup
  #...
end
```
Class Activation Schemes

• How can we ensure that a class M is active when running code from class C?

Class-based Activation
• Control flow passes through class M
• For every class c that is visited on the way to C: $c \in \text{scope}(M)$
• M pushes modifications to C (cf. local rebinding/dynamic scoping)

Mixin-based Activation
• M is a module/mixin
• C includes M
• C requests modifications from M

```
include NoPopup
class ::Viewer; end
class ::WebPage; ...
end
```
Hierarchical Scoping

• How do we share modifications among an entire class nesting hierarchy?
• Modifications of class C should affect all classes that are nested inside C
Example 2: Overview

Object  \{Object, AddressBook, AddressBook.Address, Networking, Networking.Address, String\}

AddressBook  \{AddressBook, Address, String\}

Address  \{Address\}

String  \to\_address\}

Networking  \{Networking, Address, String\}

String  \{String\}

String  \{String\}

Network  \to\_address\}

Pinging  \ping\}

Address  \to\_address\}

Address  \to\_address\}

Address  \to\_address\}
Activation / Deactivation Rule

• Extend both rules

• Activation: When calling a method \texttt{C.foo}, activate \texttt{C} and all of its enclosing classes

• Deactivation: Extend scope(\texttt{C}) such that it also includes the scope of all nested classes of \texttt{C}
Example 2: Scope of Classes

- Modifications in `Object` are globally visible
- Modifications in `AddressBook` are visible in `AddressBook` and its nested classes
- Modifications in `Networking` are visible in `Networking` and its nested classes
Example 2: Invocation Code

```ruby
module Networking
  module Pinging
    def self.ping(addr)
      addr.to_address
    end
  end
end

class Application
  def run
    Networking::Pinging.ping("127.0.0.1")
  end
end
```
Example 2: Step by Step

\[
S = \{ \text{Object} \}
\]
Example 2: Step by Step

\[
S = \{ \text{Object, Application} \}
\]
Example 2: Step by Step

S = \{ \text{Object, Networking, Pinging} \}
Example 2: Step by Step

\[ S = \{ \text{Object, Networking, String} \} \]

- **Pinging.ping**
- **String.to_address**
- **Networking remains active**

Application.run

String ∈ scope(Networking)

Run to_address from Networking
Implementation

• Prototypical implementation using metaprogramming

• Uses debug_inspector API for stack walking to implement customized method lookup in Ruby

• Give it a try (use Ruby 2.3):
  git@github.com:matthias-springer/ruby-class-ext.git
Related Work

• **Classboxes** [Bergel03]: Additional organizational unit (classbox), no support class nesting hierarchies
• **Ruby Refinements:**
  Pure lexical scoping (no local rebinding)
• **Context-oriented Programming (COP)** [Hirschfeld08]:
  Manual activation/deactivation necessary, difficult to control when modifications should be deactivated
• **Method Shells** [Takeshita13]: Additional organizational unit (method shell), new syntax for including/linking
• **MultiJava** [Clifton00], **Expanders** [Warth06]:
  No dynamic scoping, no new methods
Summary

“Extension Classes”: A new approach for open classes in Ruby

• Avoiding destructive modifications
• Reusable modifications (via modules)
• Scoped with respect to class nesting hierarchies
• Classes as only organizational unit
References


Appendix
module NoPopup
  refine WebPage do
    def popup; end
  end
end

class Browser
  using NoPopup

  def open(url)
    WebPage.new.open(url)
  end
end

Pure lexical scoping: NoPopup will be deactivated after calling WebPage.open
Combining Modifications

• What happens if multiple classes with variation points for the same method are active?

• S is actually not a set but a stack → *Class composition stack*  
  (cf. layer composition stack in COP [Hirschfeld08])

• Last activated class takes precedence

• Modified super keyword to navigate 3 hierarchies
  • Inheritance hierarchy of layer class (i.e., of class containing modifications) → takes care of mixins
  • Class composition stack (proceed in COP, AOP)
  • Inheritance hierarchy of receiver class
Method Lookup

1. Superclass inheritance hierarchy of layer class
2. Layer composition stack
3. Receiver class inheritance hierarchy
Example 3: Overview

Object {Object, AST, Nodes, Node, IntNode, PlusNode, Application}

AST {AST, Nodes, Node, IntNode, PlusNode}

Nodes {Nodes, Node, IntNode, PlusNode}

Evaluating {Evaluating, Node, IntNode, PlusNode (+ super)}

Application include AST::Evaluating include AST::Printing

Printing {Printing, Node, IntNode, PlusNode (+ super)}

Node {Node}

IntNode < Node {IntNode (+, Node)}

PlusNode < Node {PlusNode (+, Node)}

Node

IntNode evaluate

PlusNode evaluate

Node

IntNode print

PlusNode print

第110回プログラミング研究発表会 (SWoPP2016)
Example 3: Mixins

• Conceptually, module inclusion (mixin application) creates a new superclass

• Formalism does not have a special rule for mixin application, but only for superclasses
  → Assume that modules have been desugared to explicit superclasses from now on
Example 3: Inheritance

class Application
  include AST::Evaluating
  include AST::Printing

  def evaluate(node)
    return node.evaluate
  end
end

Mixins are desugared to explicit superclasses
Ex. 3: Method Lookup for \texttt{IntNode}

Mixins are desugared to explicit superclasses

Assuming single active class \texttt{Application}

- Activation/deactivation rules remain unchanged
- \textit{Effective superclass hierarchy} determines method to be executed (and also guides the lookup for \texttt{proceed\ (super)})
**Definition.** The effective superclass hierarchy of a class $C$ is defined as $\text{Effective}(C)$, where $S$ is the class composition stack ($S[1]$ is top of stack), $\#C$ is the number of superclasses of a class $C$, $\text{super}^i(C)$ is the $i$-th superclass of class $C$, $L[C]$ is the partial class targeting $C$ defined in $L$ (if there is one), $\langle \rangle$ brackets denote a (ordered) list, and summation is used for list concatenation.

\[
\text{LayerHierarchy}(L, C) = \sum_{i=0}^{\#L} \langle \text{super}^i(L)[C] \rangle
\]

\[
\text{ClassLayers}(C) = \left( \sum_{i=1}^{\#S} \text{LayerHierarchy}(S[i], C) \right) + \langle C \rangle
\]

\[
\text{Effective}(C) = \sum_{i=0}^{\#C} \text{ClassLayers}(\text{super}^i(C))
\]

LayerHierarchy($L, C$) is the list of partial classes for class $C$ defined in class $L$ and its superclasses. ClassLayers($C$) is the list of partial classes of $C$ (among all activated classes) and $C$ itself.

In Example 3:

- $S = \langle \text{Application} \rangle$ (Object omitted)
- $\text{LayerHierarchy}(\text{Application, IntNode}) = \langle \text{Application} \rangle [\text{IntNode}, \text{Evaluating()} \text{[IntNode]}, \text{Printing()} \text{[IntNode]}, \text{Object}[\text{IntNode}])$
- Same as above plus $\text{IntNode}$ (only one layer/active class, i.e., Application in this example)
- Account for superclasses of $\text{IntNode}$
Definition of “Scope of a Class”

**Definition.** The scope of a class \( L \) is defined as the set containing \( L \)

\[
\text{scope}(L) = \{L\} \quad \text{(reflexivity)}
\]

Direct method calls
Definition of “Scope of a Class”

**Definition.** The scope of a class $L$ is defined as the set containing $L$, all target classes corresponding to partial classes of $L$.

$$\text{scope}(L) = \{L\} \cup \{\text{target}(P) \mid P \in \text{partials}(L)\}$$

*Indirect method calls* (reflexivity) *(dynamic scoping + local rebinding)*
Definition of “Scope of a Class”

Definition. The scope of a class \( L \) is defined as the set containing \( L \), all target classes (and their reachable nested classes\(^6\)) corresponding to partial classes of \( L \), all classes in the scope of all nested classes of \( L \)\(^7\).

\[
\text{scope}(L) = \{L\} \quad \text{(reflexivity)}
\]

\[
\cup \{C \mid C \in \text{nested}(\text{target}(\text{P})) \land P \in \text{partials}(L)\} \quad \text{(dynamic scoping + local rebinding (+ hierarch. scoping))}
\]

\[
\cup \{C \mid C \in \text{scope}(N) \land N \in \text{nested}(L)\} \quad \text{(hierarch. scoping)}
\]

Visible in all nested classes of target classes

Visible in all nested classes of defining class
**Definition of “Scope of a Class”**

**Definition.** The scope of a class \( L \) is defined as the set containing \( L \), all target classes (and their reachable nested classes\(^*6\)) corresponding to partial classes of \( L \), all classes in the scope of all nested classes of \( L \)*\(^7\), and all classes in the scope of the superclass of \( L \)

\[
\text{scope}(L) = \{L\} \quad \text{(reflexivity)}
\]

\[
\bigcup \{C \mid C \in \text{nested}(\text{target}(P)) \land P \in \text{partials}(L)\} \quad \text{(dynamic scoping + local rebinding (+ hierarch. scoping))}
\]

\[
\bigcup \{C \mid C \in \text{scope}(\text{nested}(L))\} \quad \text{(hierarch. scoping)}
\]

\[
\bigcup \text{scope}(\text{superclass}(L)) \quad \text{(inheritance scoping)}
\]

\(^{6}\)Visible in nested classes + target classes of superclasses
Definition of “Scope of a Class”

**Definition.** The scope of a class $L$ is defined as the set containing $L$, all target classes (and their reachable nested classes*) corresponding to partial classes of $L$, all classes in the scope of all nested classes of $L^*$, and all classes in the scope of the superclass of $L$ (if super($L$) ≠ Object).

$$
\text{scope}(L) = \cup \{C \mid C \in \text{nested}^*(\text{target}(P)) \land P \in \text{partials}(L)\}
$$

(dynamic scoping + local rebinding (+ hierarch. scoping))

$$
\cup \{C \mid C \in \text{scope}(N) \land N \in \text{nested}(L)\} \quad \text{(hierarch. scoping)}
$$

$$
\cup \text{scope}(\text{superclass}(L)) \quad \text{(inheritance scoping)}
$$

* Special rule because Object is the superclass of all classes. scope(Object) contains all classes.